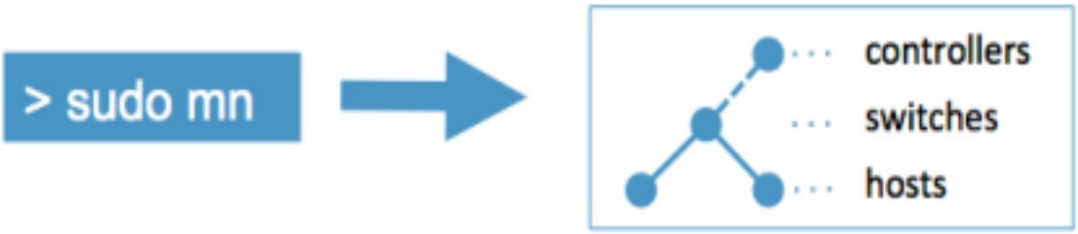
Mininet 应用与源码剖析



yeasy@github

Table of Contents

Mininet 应用与源码剖析

Mininet 是研究软件定义网络，进行快速验证的高效模拟平台。

本书的一到三章介绍 Mininet 的安装和使用；第四章介绍一些高级功能；五到七章分析 Mininet 的源码实现。

最新版本在线阅读：GitBook。

本书源码在 Github 上维护，欢迎参与： https://github.com/yeasy/mininet_book。

感谢所有的 贡献者。

更新历史:

- 0.6: 2013-12-16
  - 完善对 topo 模块、net 模块等的分析。
- 0.5: 2013-12-11
  - 补充对部分 example 文件的分析；
  - 增加对 clean 模块的分析。
- 0.4: 2013-11-18
  - 补充对 link 和 node 库的核心代码分析； 补 充运行文件分析。
- 0.3: 2013-11-16
  - 完成运行文件分析。
- 0.2: 2013-11-15
  - 完成库文件分析。
- 0.1: 2013-10-11
  - 完成代码结构分析。

参加步骤

- 在 GitHub 上 `fork` 到自己的仓库，如 `user/mininet_book`，然后 `clone` 到本地，并设置用户信息。

```
$ git clone git@github.com:user/mininet_book.git
$ cd mininet_book
$ git config user.name "User"
$ git config user.email user@email.com
```

- 修改代码后提交，并推送到自己的仓库。

```
$ #do some change on the content
$ git commit -am "Fix issue #1: change helo to hello"
$ git push
```

- 在 GitHub 网站上提交 pull request。
- 定期使用项目仓库内容更新自己仓库内容。 ``` $ git remote add upstream https://github.com/yeasy/mininet_book $ git fetch upstream $ git checkout master $ git rebase upstream/master $ git push -f origin master

概述

Stanford 大学 Nick McKeown 教授领导的研究小组基于 Linux Container 架构，开发出了这套进程虚拟化的平台。在 Mininet 的帮助下，你可以轻易的在自己的笔记本上测试一个软件定义网络（Software-Defined Networks），对基于 Openflow、Open vSwitch 的各种协议等进行开发验证，或者验证自己的想法。

最令人振奋的是，所有的代码几乎可以无缝迁移到真实的硬件环境中。在实验室里，一行命令就可以创建一个支持 SDN 的任意拓扑的网络结构，并可以灵活的进行相关测试，验证了设计的正确后，又可以轻松部署到真实的硬件环境中。目前 Mininet 已经作为官方的演示平台对各个版本的 Openflow 协议进行演示和测试。

Mininet 项目的首页在 http://www.openflowswitch.org/foswiki/bin/view/OpenFlow/Mininet，目前主要维护者为 Bob Lantz 跟 Brandon Heller。讨论组地址为 mininet-discuss@lists.stanford.edu。

主要特性

Mininet 作为一个轻量级软定义网络研发和测试平台，其主要特性包括:

- 支持 Openflow、OpenvSwitch 等软定义网络部件
- 方便多人协同开发
- 支持系统级的还原测试支持复杂拓扑、自定义拓扑
- 提供 Python API
- 很好的硬件移植性（Linux 兼容），结果有更好的说服力
- 高扩展性，支持超过 4096 台主机的网络结构

安装

使用镜像

下载

官方网站已经提供了配置好相关环境的基于 Debian Lenny 的虚拟机镜像，下载地址为http://openflowswitch.org/downloads/OpenFlowTutorial-081910.vmware.zip， 压缩包大小为 700 MB 左右，解压后大小为 2.1 GB 左右。虚拟机镜像格式为 Vmware 的 vmdk，可以直接使用 vmware workstation 或者 Virtualbox 等软件打开。如果使用 QEMU 和 KVM 则需要先进行格式转换。

如果使用 Virtualbox 进行加载，需要注意 尽量使用最新版本，host 操作系统需要支持 PAE，并在 Virtualbox 中打开 PAE 支持。

使用

默认用户名密码均为 openflow，建议通过本地利用 ssh 登录到虚拟机上使用（可以设置自动登录并将 X 重定向到本地），比较方便操作。

注意事项： 建议将 guest 主机采用 bridge 方式联网，以获取 host 可见的独立 IP；也可采用为 guest 配置两块网卡方式，一块采用 NAT（一般来说，guest 上看到为 eth0，IP 地址为 `10.0.2.*`，网关为 `10.0.2.2`），一块采用 host-only（guest 上的 eth1，IP 地址为`192.168.56.*`），但 host-only 的网卡可能无法自动 dhcp 到地址，需要手动配置（`ifconfig eth1 ip/mask`）将 host 机 .ssh 目录下 id_rsa.pub 复制到 guest 机的 .ssh 目录下，并写入 authorized_keys，可实现自动登陆认证。

本地安装

大部分发行版中已经带有该软件包，直接通过命令安装即可。例如，在 Ubuntu 系统中，执行

```
$ sudo aptitude install -y mininet
```

也可以通过源代码安装。

```
$ git clone https://github.com/mininet/mininet.git
$ cd mininet
```

参考 INSTALL 文件中针对不同操作系统的安装步骤。

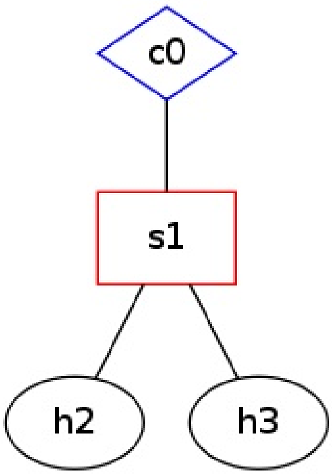
基本使用

通过几个简单例子来介绍 Mininet 的基本功能。

创建网络

Mininet 的操作十分简单，启动一个小型测试网络只需要下面几个步骤。 登录到虚拟机命令行界面，打开 wireshark，使其后台运行, 命令为 `sudo wireshark &`。

启动 Mininet，命令为 `sudo mn`，则默认创建如下图所示的网络拓扑。



经过短暂的等待即可进入以 `mininet>` 引导的命令行界面。 好了，从现在开始，我们就拥有了一个 1 台控制节点(controller)、一台交换(switch)、两台主机(host)的网络，并且用 wireshark 进行观测。

查看信息

查看全部节点：

```
mininet> nodes
available nodes are:
c0 h2 h3 s1
```

查看链路信息：

```
mininet> net
s1 <-> h2-eth0 h3-eth0
```

输出各节点的信息：

```
mininet> dump
c0: IP=127.0.0.1 intfs= pid=1679
s1: IP=None intfs=s1-eth1,s1-eth2 pid=1682
h2: IP=10.0.0.2 intfs=h2-eth0 pid=1680
h3: IP=10.0.0.3 intfs=h3-eth0 pid=1681
```

对节点进行单独操作

如果想要对某个节点的虚拟机单独进行命令操作，也十分简单，命令格式为 `node cmd` 。

例如查看交换机 s1 上的网络信息，我们只需要在执行的 ifconfig 命令前加上 s1 主机标志即可，即 `s1 ifconfig` ，同样，如果我们想用 ping 3 个包的方法来测试 h2 跟 h3 之间连通情况，只需要执行 `h2 ping -c 3 h3` 即可。得到的结果为

```
mininet> h2 ping -c 3 h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=7.19 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.239 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.136 ms
— 10.0.0.3 ping statistics —
3 packets transmitted, 3 received, 0% packet loss, time 2006ms
rtt min/avg/max/mdev = 0.136/2.523/7.194/3.303 ms
```

在本操作执行后，可以通过 wireshark 记录查看到创建新的流表项的过程，这也是造成第一个 ping 得到的结果偏大的原因。

更简单的全网络互 ping 测试命令是 `pingall` ，会自动所有主机节点逐对进行 ping 连通测试。

常用操作

介绍一些常用的操作命令。

快捷测试

除了 cli 的交互方式之外，Mininet 还提供了更方便的自动执行的快捷测试方式，其格式为 `sudo mn --test cmd`，即可自动启动并执行 cmd 操作，完成后自动退出。

例如 `sudo mn --test pingpair`，可以直接对主机连通性进行测试，`sudo mn --test iperf` 启动后直接进行性能测试。用这种方式很方便直接得到实验结果。

自定义拓扑

Mininet 提供了 Python API，可以用来方便的自定义拓扑结构。

在 mininet/custom 目录下给出了几个例子。例如在 topo-2sw-2host.py 文件中定义了一个 mytopo，则可以通过 `--topo` 选项来指定使用这一拓扑，命令为 `sudo mn --custom ~/mininet/custom/topo-2sw-2host.py --topo mytopo --test pingall`。

同样的，我们可以通过下面的 Python 脚本来完成对一个2 层 tree 拓扑网络的测试。

```
from mininet.net import Mininet
from mininet.topolib import TreeTopo
tree4 = TreeTopo(depth=2,fanout=2)
net = Mininet(topo=tree4)
net.start()
h1, h4 = net.hosts[0], net.hosts[3]
print h1.cmd('ping -c1 %s' % h4.IP())
net.stop()
```

使用友好的 MAC 编号

默认情况下，主机跟交换机启动后分配的 MAC 地址是随机的，这在某些情况下不方便查找问题。

可以使用 `--mac` 选项，这样主机跟交换机分配到的 MAC 地址跟他们的 ID 是一致的，容易通过 MAC 地址较快找到对应的节点。

使用XTerm

为了能够正确使用 xterm，我们需要做些准备工作。在这里推荐利用远程方式登录到 OpenflowVM。

客户端

对于自带 X 的 Linux 主机，无需配置 X。

如果客户端是 Windows 主机，需要先在 Windows 机器上安装 Xserver（Xming）。下载地址：http://sourceforge.net/projects/xming/files/Xming/6.9.0.31/Xming-6-9-0-31-setup.exe。

如果你使用 secureCRT 远程登录到 OpenflowVM，需要在“会话选项->SSH2->密钥交换”下，取消 `diffie-hellman-group14` 和 `diffie-hellman` 选择；同时在“远程/X11”下，选择转发“X11数据包（F）”；点击确定。

开启 Xming，使用如下命令远程到Openflow VM即可。

```
ssh -X openflow@[Guest IP here]
```

如果你使用 puTTY 远程登录到 OpenflowVM: 点击“puTTY->Connection->SSH->X11”，选择“X11 forwarding->Enable X11 forwaring”；开启 Xming，点击 Windows 开始按钮，在运行栏输入“cmd”，打开终端，输入 `cd <dir>` 切换到保存 puTTY 的目录下；使用下面的命令远程登录到 openflow VM。

```
putty.exe -X openflow@[Guest IP here]
```

主机端

通过使用 `-x` 参数，Mininet 在启动后会在每个节点上自动打开一个 XTerm，方便某些情况下的对多个节点分别进行操作。命令为

```
sudo mn -x
```

在进入 mn cli之后，也可以使用 xterm node 命令指定启动某些节点上的 xterm，例如分别启用 s1 跟 h2 上的 xterm，可以用

```
xterm s1 h2
```

链路操作

在 Mininet cli 中，使用 `link` 命令，禁用或启用某条链路，格式为

```
link node1 node2 up/down
```

例如临时禁用 s1 跟 h2 之间的链路，可以用

```
link s1 h2 down
```

指定交换机跟控制器类型

通过 `--switch` 选项跟 `--controller` 选项可以分别指定采用哪种类型的交换机跟控制器。

例如使用用户态的交换机：

```
sudo mn --switch user
```

使用 OpenvSwitch：

```
sudo mn --switch ovsk
```

使用 NOX pyswitch：

- 首先确保 NOX 运行

  ```
  cd $NOX_CORE_DIR
  ./nox_core -v -i ptcp:
  ```

  然后 `Ctrl-c` 杀死 NOX 进程
- 然后指定 NOX 交换机

  ```
  sudo -E mn --controller nox_pysw
  ```

注意：通过`-E`选项来保持预定义的环境变量（此处为 `NOX_CORE_DIR`）。

名字空间

默认情况下，主机节点有用独立的名字空间（namespace），而控制节点跟交换节点都在根名字空间（root namespace）中。

如果想要让所有节点拥有各自的名字空间，需要添加 `--innamespace` 参数，即启动方式为 `sudo mn --innamespace`。

注意：为了方便测试，在默认情况下，所有节点使用同一进程空间，因此，在 h2 跟 h3 或者 s1 上使用 `ps` 查看进程得到的结果是一致的，都是根名字空间中的进程信息。

启动参数总结

- `-h, --help` 打印帮助信息
- `--switch=SWITCH` 交换机类型，包括 [kernel user ovsk]
- `--host=HOST` 模拟主机类型，包括 [process]
- `--controller=CONTROLLER` 控制器类型，包括 [nox_dump none ref remote nox_pysw]
- `--topo=TOPO,arg1,arg2,...argN` 指定自带拓扑，包括 [tree reversed single linear minimal]
- `-c, --clean` 清理环境
- `--custom=CUSTOM` 使用自定义拓扑和节点参数
- `--test=TEST` 测试命令，包括 [cli build pingall pingpair iperf all iperfudp none]
- `-x, --xterms` 在每个节点上打开 xterm
- `--mac` 让MAC 地址跟 DP ID 相同
- `--arp` 配置所有 ARP 项
- `-v VERBOSITY, --verbosity=VERBOSITY [info warning critical error debug output]` 输出日志级别
- `--ip=IP` 远端控制器的IP地址
- `--port=PORT` 远端控制器监听端口
- `--innamespace` 在独立的名字空间内
- `--listenport=LISTENPORT` 被动监听的起始端口
- `--nolistenport` 不使用被动监听端口
- `--pre=PRE` 测试前运行的 CLI 脚本
- `--post=POST` 测试后运行的 CLI 脚本

常用命令总结

- `help` 默认列出所有命令文档，后面加命令名将介绍该命令用法
- `dump` 打印节点信息
- `gterm` 给定节点上开启 gnome-terminal。注：可能导致 Mininet 崩溃
- `xterm` 给定节点上开启 xterm
- `intfs` 列出所有的网络接口
- `iperf` 两个节点之间进行简单的 iperf TCP测试
- `iperfudp` 两个节点之间用指定带宽 udp 进行测试
- `net` 显示网络链接情况
- `noecho` 运行交互式窗口，关闭回应（echoing）
- `pingpair` 在前两个主机之间互 ping 测试
- `source` 从外部文件中读入命令
- `dpctl` 在所有交换机上用 dptcl 执行相关命令，本地为 `tcp 127.0.0.1:6634`
- `link` 禁用或启用两个节点之间的链路
- `nodes` 列出所有的节点信息
- `pingall` 所有 host 节点之间互 ping
- `py` 执行 Python 表达式
- `sh` 运行外部 shell 命令
- `quit/exit` 退出

其他操作

执行 `sudo mn -c` 会进行清理配置操作，适合故障后恢复。 执行 `exit` 会退出 Mininet 的 CLI，同时给出运行时间统计。 `py cmd` 使用 Python 来执行 cmd。 测试 Mininet 启动后立刻关闭的时间可以用 `sudo mn --test none`。

高级功能

本章将通过一个具体管理 Openflow switch 的例子来介绍一些比较高级的命令。

首先，启动 Mininet，执行

```
sudo mn --topo single,3 --mac --switch ovsk --controller remote
```

生成一个小的网络，三台主机连到一台交换机上，交换机为 OpenvSwitch 交换机，指定 remote 类型控制器（默认为本地）。

使用 **dpctl**

执行

```
dpctl show tcp:127.0.0.1:6634
```

可以查看到交换机的端口等基本情况，其中 tcp 端口 6634 是默认的交换机监听端口。

执行

```
dpctl dump-flows tcp:127.0.0.1:6634
```

可以看到更详细的流表信息。 此时，流表为空，执行 `h2 ping h3` 无法得到响应。因此我们需要通过 dpctl 手动添加流表项，实现转发。 命令为

```
dpctl add-flow tcp:127.0.0.1:6634 in_port=1,actions=output:2
dpctl add-flow tcp:127.0.0.1:6634 in_port=2,actions=output:1
```

此时查看流表可以看到新的转发信息，同时可以在 h2 和 h3 之间 ping 通。

控制器

通过执行

```
sudo mn --controller=remote --ip=[controller IP] --port=[controller listening port]
```

可以连接到控制器。

交换机与控制器交互

我们可以启动一个简单的控制器，默认没有任何流表项，仅仅作为一台带学习功能的交换机。控制器默认监听端口是 6633。

以下控制器与交换机之间的消息交互过程，可以通过 wireshark，配置 of 过滤器观察到交换机跟控制器之间的交互消息。 参见下面的表格。

消息	类型	描述
Hello	Controller->Switch	跟着 TCP 握手，控制器发送它的版本号到交换机。
Hello	Switch->Controller	交换机回复它支持的版本。
Features Request	Controller->Switch	控制器询问可用端口。
Set Config	Controller->Switch	控制器让交换机发送流超时消息。
Features Reply	Switch->Controller	交换机回复端口、端口速度、支持的表和行动。

同样，我们可以用 wireshark 观察到当第一次有 ping 包从 h2 发到 h3 时，控制器如何自动添加相应的表项到交换机。wireshark 相应的过滤器为 `of && (of.type != 3) && (of.type != 2)`。

相关的消息过程参考下面的表格。 | 消息 | 类型 | 描述 | | -- | -- | -- | | Packet-In | Switch->Controller | 网包到达交换机，没有发生匹配，发送到控制器。| | Packet-Out | Controller->Switch | 从交换机端口发出包。| | Flow-Mod | Controller->Switch | 添加指定流。| | Flow-Expired | Switch->Controller | 流超时。|

使用 NOX

首先确定没有其他控制器在运行（占据 6633 端口）

```
sudo killall controller
```

启动 Mininet

```
sudo mn --topo single,3 --mac --switch ovsk --controller remote
```

然后启动 NOX，默认路径为 ~/noxcore/build/src，重新打开一个 ssh 终端执行

```
./nox_core -v -i ptcp: pytutorial
```

会自动打开运行 tutorial 应用的 NOX，打印出详细的调试信息，并监听 6633 端口。

直到打印出类似如下信息，说明交换机已经成功连接到 NOX。

```
00039|nox|DBG:Registering switch with DPID = 1
```

通过互 ping 测试，各个主机连通，此时 switch 等同于一个 hub。 然后通过修改 ~/noxcore/src/nox/coreapps/tutorial/pytutorial.py 中代码，让 NOX 工作成一个带学习功能的交换机。相关命令参考 ofinclude 代码，以及 NOX 对各个包的解析代码目录：~/noxcore/src/nox/lib/packet/。

通过编写 NOX 程序，我们可以让交换机的行为更加智能化、复杂化。为了测试我们编写的 NOX 程序，我们可以使用 cbench 来进行测试。

外部读取配置命令

可以写到一个文件中，用 Mininet 直接调用。例如脚本文件名为 my_cli_script，则可以执行

```
mininet> source my_cli_script
```

或者

```
# mn --pre my_cli_script
```

代码结构

介绍源代码的基本结构。

运行相关

- `bin/mn` 主运行文件，安装后执行 mn 即调用的本程序，是 Python 程序。
- `mnexec.c` 执行一些快速命令，比如关闭文件描述符等，是 C 程序，编译后生成二进制文件 mnexec 被 Python 库调用。

安装相关

- INSTALL：安装说明
- setup.py：安装 Python 包时候的配置文件，被 Makefile 中调用。
- debian/：生成 deb 安装包时的配置文件。

核心代码

核心代码基本都在 mininet/ 子目录下。 注：最新的 2.1.0 版本，核心 Python 代码仅为 4675 行。

```
$ find mininet  -name "*.py" | xargs cat | wc -l
```

说明文件

- CONTRIBUTORS：作者信息
- README.md：主说明文件
- doc/doxygen.cfg：执行doxygen生成文档时的配置文件。

其它

- LICENSE
- custom/ 目录下可以放一些用户自定义的 Python 文件，比如自定义的拓扑类等。
- test/ 目录下是一些测试的例子。

util/

目录下是一些辅助文件，包括安装脚本、文档辅助生成等，重要的文件包括：

- m bash 脚本提供用户直接在 host 执行命令的接口。例如

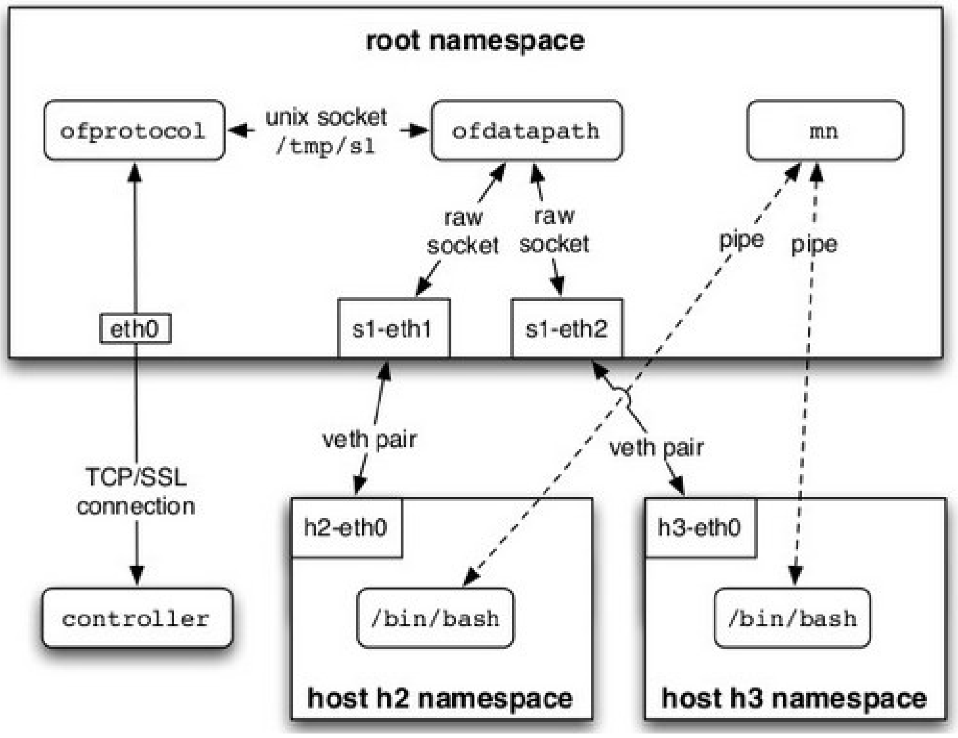
  ```
  m host cmd args...
  ```

  m 通过调用 mnexec 来实现对 Mininet 中的元素执行相应的命令。
- mnexec C 程序，通过参数绑定到某个名字空间，并执行给定的命令。

整体逻辑功能

整体上来看，Mininet 作为一个基于 Python 的网络模拟工具，可以分为两大部分：Python库和运行文件。

前者提供对网络中元素进行抽象和实现，例如定义主机类来表示网络中的一台主机。后者则基于这些库来完成各种自定义的模拟过程。 一个典型的场景如下图所示。



mininet.link 模块

描述链路相关的接口和连接。包括 Intf 类、Link 类、TCIntf 类和 TCLink 类。

mininet.link.Intf

表示基本的网络接口，比如 h1-eth0 表示 host 1 上的 eth0 接口。 属性包括所在的节点，名称，所接的link，mac/ip 信息等。 构造的时候会传入节点、端口等属性，并绑定接口到对应的节点的端口上。

```
def __init__( self, name, node=None, port=None, link=None, **params ):
        """name: interface name (e.g. h1-eth0)
           node: owning node (where this intf most likely lives)
           link: parent link if we're part of a link
           other arguments are passed to config()"""
        self.node = node
        self.name = name
        self.link = link
        self.mac, self.ip, self.prefixLen = None, None, None
        # Add to node (and move ourselves if necessary )
        node.addIntf( self, port=port )
        # Save params for future reference
        self.params = params
        self.config( **params )
```

所支持的方法包括配置 mac/ip 等配置方法，大都是通过 ifconfig 命令在对应节点上调用cmd方法来实现。此外，还提供了 config() 方法来一次性配置所有的属性。

mininet.link.Link

表示基本的一条链路，最基本的链路在 mininet 中其实就是一对 veth 接口对。

```
def __init__( self, node1, node2, port1=None, port2=None,
              intfName1=None, intfName2=None,
              intf=Intf, cls1=None, cls2=None, params1=None,
              params2=None ):
    """Create veth link to another node, making two new interfaces.
       node1: first node
       node2: second node
       port1: node1 port number (optional)
       port2: node2 port number (optional)
       intf: default interface class/constructor
       cls1, cls2: optional interface-specific constructors
       intfName1: node1 interface name (optional)
       intfName2: node2  interface name (optional)
       params1: parameters for interface 1
       params2: parameters for interface 2"""
    # This is a bit awkward; it seems that having everything in
    # params would be more orthogonal, but being able to specify
    # in-line arguments is more convenient!
    if port1 is None:
        port1 = node1.newPort()
    if port2 is None:
        port2 = node2.newPort()
    if not intfName1:
        intfName1 = self.intfName( node1, port1 )
    if not intfName2:
        intfName2 = self.intfName( node2, port2 )

    self.makeIntfPair( intfName1, intfName2 )

    if not cls1:
        cls1 = intf
    if not cls2:
        cls2 = intf
    if not params1:
        params1 = {}
    if not params2:
        params2 = {}

    intf1 = cls1( name=intfName1, node=node1, port=port1,
                  link=self, **params1  )
    intf2 = cls2( name=intfName2, node=node2, port=port2,
                  link=self, **params2 )

    # All we are is dust in the wind, and our two interfaces
    self.intf1, self.intf2 = intf1, intf2
```

创建链路时，需要在两个节点上分别生成两个端口，利用节点和端口，获取对应的两个网络接口的名称，例如 s1-eth0 和 h1-eth0，然后调用 makeIntfPair() 方法，最终调用 util.py 中的 makeIntfPair() 方法，调用系统中的 `ip link` 命令来创造一对 veth pair。

```
def makeIntfPair( intf1, intf2 ):
    """Make a veth pair connecting intf1 and intf2.
       intf1: string, interface
       intf2: string, interface
       returns: success boolean"""
    # Delete any old interfaces with the same names
    quietRun( 'ip link del ' + intf1 )
    quietRun( 'ip link del ' + intf2 )
    # Create new pair
    cmd = 'ip link add name ' + intf1 + ' type veth peer name ' + intf2
    return quietRun( cmd )
```

mininet.link.TCIntf

被 TC（Linux 下的 traffic control 的工具）自定义的接口，可以配置包括带宽、延迟、丢包率、最大队列长度等参数。

mininet.link.TCLink

表示一对对称的 TC 接口连接到一起。

mininet.node 模块

节点模块表示网络中的基本元素（包括主机、交换机和控制器），十分关键。

其中，每个主机默认在一个单独的名字空间中，交换机和控制器都在 root 名字空间中。

基类

即 mininet.node.Node， 表示一个基本的虚拟网络节点，是所有的网络节点的父类。

实现上其实就是在网络名字空间中的一个shell进程，可以通过各种管道进行通信。该类是模块中其他类的根本，其它类都是直接或间接继承。

节点包括名称、是否在网络名字空间、接口、端口等可能的属性。

init

```
params: Node parameters (see config() for details)"""

    # Make sure class actually works
    self.checkSetup()

    self.name = name
    self.inNamespace = inNamespace

    # Stash configuration parameters for future reference
    self.params = params

    self.intfs = {}  # dict of port numbers to interfaces
    self.ports = {}  # dict of interfaces to port numbers
                     # replace with Port objects, eventually ?
    self.nameToIntf = {}  # dict of interface names to Intfs

    # Make pylint happy
    ( self.shell, self.execed, self.pid, self.stdin, self.stdout,
        self.lastPid, self.lastCmd, self.pollOut ) = (
            None, None, None, None, None, None, None, None )
    self.waiting = False
    self.readbuf = ''

    # Start command interpreter shell
    self.startShell()
```

初始化函数主要进行参数的初始化，之后通过调用 startShell() 启动一个 shell进程（该进程默认关闭描述符，并从 tty 上分离开来），等待接受传入的命令。句柄被发送给 self.shell 上。

addIntf

```
def addIntf( self, intf, port=None ):
        """Add an interface.
           intf: interface
           port: port number (optional, typically OpenFlow port number)"""
        if port is None:
            port = self.newPort()
        self.intfs[ port ] = intf
        self.ports[ intf ] = port
        self.nameToIntf[ intf.name ] = intf
        debug( '\n' )
        debug( 'added intf %s:%d to node %s\n' % ( intf, port, self.name ) )
        if self.inNamespace:
            debug( 'moving', intf, 'into namespace for', self.name, '\n' )
            moveIntf( intf.name, self )
```

添加一个接口（比如 `<Intf h1-eth0>` ）到节点上，如果给定了 port（比如 0 ），则建立端口到接口的映射关系。这个映射关系通过 self.intfs 和 self.ports 两个字典来分别维护。

cmd

该函数能在节点所在的进程shell上执行输入的命令。

```
def cmd( self, *args, **kwargs ):
        """Send a command, wait for output, and return it.
           cmd: string"""
        verbose = kwargs.get( 'verbose', False )
        log = info if verbose else debug
        log( '*** %s : %s\n' % ( self.name, args ) )
        self.sendCmd( *args, **kwargs )
        return self.waitOutput( verbose )
```

config

配置 MAC，IP 或 default Route 信息。

```
def config( self, mac=None, ip=None,
                defaultRoute=None, lo='up', **_params ):
        """Configure Node according to (optional) parameters:
           mac: MAC address for default interface
           ip: IP address for default interface
           ifconfig: arbitrary interface configuration
           Subclasses should override this method and call
           the parent class's config(**params)"""
        # If we were overriding this method, we would call
        # the superclass config method here as follows:
        # r = Parent.config( **_params )
        r = {}
        self.setParam( r, 'setMAC', mac=mac )
        self.setParam( r, 'setIP', ip=ip )
        self.setParam( r, 'setDefaultRoute', defaultRoute=defaultRoute )
        # This should be examined
        self.cmd( 'ifconfig lo ' + lo )
        return r
```

connectionsTo

返回所有从自身连接到给定节点的接口，即 [intf1, intf2...]。

```
def connectionsTo( self, node):
        "Return [ intf1, intf2... ] for all intfs that connect self to node."
        # We could optimize this if it is important
        connections = []
        for intf in self.intfList():
            link = intf.link
            if link:
                node1, node2 = link.intf1.node, link.intf2.node
                if node1 == self and node2 == node:
                    connections += [ ( intf, link.intf2 ) ]
                elif node1 == node and node2 == self:
                    connections += [ ( intf, link.intf1 ) ]
        return connections
```

主机类

包括 Host 和 CPULimitedHost 两个类。

mininet.node.Host

表示一个主机节点，目前跟 Node 类定义相同。 在主机类上执行命令可以通过 Cmd() 或者 sendCmd() 方法，前者会等待命令的输出结果，后者会直接返回，并允许使用后续的 monitor() 来进行监视跟踪。

mininet.node.CPULimitedHost

继承自 Host 类，通过 cgroup 工具来对 CPU 进行限制。

init

```
def __init__( self, name, sched='cfs', **kwargs ):
        Host.__init__( self, name, **kwargs )
        # Initialize class if necessary
        if not CPULimitedHost.inited:
            CPULimitedHost.init()
        # Create a cgroup and move shell into it
        self.cgroup = 'cpu,cpuacct,cpuset:/' + self.name
        errFail( 'cgcreate -g ' + self.cgroup )
        # We don't add ourselves to a cpuset because you must
        # specify the cpu and memory placement first
        errFail( 'cgclassify -g cpu,cpuacct:/%s %s' % ( self.name, self.pid ) )
        # BL: Setting the correct period/quota is tricky, particularly
        # for RT. RT allows very small quotas, but the overhead
        # seems to be high. CFS has a mininimum quota of 1 ms, but
        # still does better with larger period values.
        self.period_us = kwargs.get( 'period_us', 100000 )
        self.sched = sched
        self.rtprio = 20
```

控制器类

mininet.node.Controller

控制器基类。默认的控制器是一个参考的实现，controller。

表示一个控制器节点。包括 IP 地址、端口等。

主要方法包括启动和停止一个控制器。

init

```
def __init__( self, name, inNamespace=False, command='controller',
                 cargs='-v ptcp:%d', cdir=None, ip="127.0.0.1",
                 port=6633, **params ):
        self.command = command
        self.cargs = cargs
        self.cdir = cdir
        self.ip = ip
        self.port = port
        Node.__init__( self, name, inNamespace=inNamespace,
                       ip=ip, **params  )
        self.cmd( 'ifconfig lo up' )  # Shouldn't be necessary
        self.checkListening()
```

checkListening

确保系统中安装了 telnet，并且在监听端口上并没有其他控制器在监听。

start

```
def start( self ):
        """Start <controller> <args> on controller.
           Log to /tmp/cN.log"""
        pathCheck( self.command )
        cout = '/tmp/' + self.name + '.log'
        if self.cdir is not None:
            self.cmd( 'cd ' + self.cdir )
        self.cmd( self.command + ' ' + self.cargs % self.port +
                  ' 1>' + cout + ' 2>' + cout + '&' )
        self.execed = False
```

该方法检查控制器程序存在，并根据传入的参数来启动它。

stop

```
def stop( self ):
        "Stop controller."
        self.cmd( 'kill %' + self.command )
        self.terminate()
```

该方法杀死控制器进程和节点的 shell 进程，并执行清理工作。

mininet.node.NOX

表示一个 NOX 控制器（需要系统中事先安装了 NOX）。

mininet.node.OVSController

表示一个ovs-controller（需要系统中实现安装了 ovs-controller）。

mininet.node.RemoteController

表示一个在 Mininet 控制外的控制器，即用户自己额外运行了控制器，此处需要维护连接的相关信息。

交换机类

mininet.node.Switch

表示一个交换机的基类。

运行在 root 名字空间。主要包括 dpid、listenport 等属性。

mininet.node.IVSSwitch

表示一台 indigo 交换机（需要系统中已存在）。

mininet.node.OVSLegacyKernelSwitch

传统的 openvswitch 交换机，基于 ovs-openflowd。不推荐。

mininet.node.OVSSwitch

表示一台 openvswitch 交换机（需要系统中已经安装并配置好 openvswitch），基于 ovs-vsctl 进行操作。目前所谓的 OVSKernelSwitch 实际上就是 OVSSwitch

mininet.node.UserSwitch

用户态的 openflow 参考交换机，即 ofdatapath。不推荐。

mininet.net 模块

主要包括 Mininet 和 MininetWithControlNet 两个类。

mininet.net.Mininet

模拟一个 Mininet 中的网络，包括拓扑、交换机、主机、控制器、链路、接口等。 其中最主要的部分是 build() 函数，依次执行：根据拓扑创建网络，配置网络名字空间，配置主机的 IP、MAC 等信息，检查是否启动 xterm，是否配置自动静态 arp 等。

init

```
  inNamespace=False,
          autoSetMacs=False, autoStaticArp=False, autoPinCpus=False,
          listenPort=None ):
"""Create Mininet object.
   topo: Topo (topology) object or None
   switch: default Switch class
   host: default Host class/constructor
   controller: default Controller class/constructor
   link: default Link class/constructor
   intf: default Intf class/constructor
   ipBase: base IP address for hosts,
   build: build now from topo?
   xterms: if build now, spawn xterms?
   cleanup: if build now, cleanup before creating?
   inNamespace: spawn switches and controller in net namespaces?
   autoSetMacs: set MAC addrs automatically like IP addresses?
   autoStaticArp: set all-pairs static MAC addrs?
   autoPinCpus: pin hosts to (real) cores (requires CPULimitedHost)?
   listenPort: base listening port to open; will be incremented for
       each additional switch in the net if inNamespace=False"""
self.topo = topo
self.switch = switch
self.host = host
self.controller = controller
self.link = link
self.intf = intf
self.ipBase = ipBase
self.ipBaseNum, self.prefixLen = netParse( self.ipBase )
self.nextIP = 1  # start for address allocation
self.inNamespace = inNamespace
self.xterms = xterms
self.cleanup = cleanup
self.autoSetMacs = autoSetMacs
self.autoStaticArp = autoStaticArp
self.autoPinCpus = autoPinCpus
self.numCores = numCores()
self.nextCore = 0  # next core for pinning hosts to CPUs
self.listenPort = listenPort

self.hosts = []
self.switches = []
self.controllers = []

self.nameToNode = {}  # name to Node (Host/Switch) objects

self.terms = []  # list of spawned xterm processes

Mininet.init()  # Initialize Mininet if necessary

self.built = False
if topo and build:
    self.build()
```

该方法主要根据传入参数配置相关的数据结构。如果还通过参数执行了拓扑信息，则执行 build 方法，调用 buildFromTopo 方法来根据拓扑信息创建节点和相关的连接等。

buildFromTopo

```
def buildFromTopo( self, topo=None ):
        """Build mininet from a topology object
           At the end of this function, everything should be connected
           and up."""

        # Possibly we should clean up here and/or validate
        # the topo
        if self.cleanup:
            pass

        info( '*** Creating network\n' )

        if not self.controllers and self.controller:
            # Add a default controller
            info( '*** Adding controller\n' )
            classes = self.controller
            if type( classes ) is not list:
                classes = [ classes ]
            for i, cls in enumerate( classes ):
                self.addController( 'c%d' % i, cls )

        info( '*** Adding hosts:\n' )
        for hostName in topo.hosts():
            self.addHost( hostName, **topo.nodeInfo( hostName ) )
            info( hostName + ' ' )

        info( '\n*** Adding switches:\n' )
        for switchName in topo.switches():
            self.addSwitch( switchName, **topo.nodeInfo( switchName) )
            info( switchName + ' ' )

        info( '\n*** Adding links:\n' )
        for srcName, dstName in topo.links(sort=True):
            src, dst = self.nameToNode[ srcName ], self.nameToNode[ dstName ]
            params = topo.linkInfo( srcName, dstName )
            srcPort, dstPort = topo.port( srcName, dstName )
            self.addLink( src, dst, srcPort, dstPort, **params )
            info( '(%s, %s) ' % ( src.name, dst.name ) )

        info( '\n' )
```

mininet.net.MininetWithControlNet

继承自 Mininet 类，主要用于在使用用户态 datapath 的时候模拟一个控制器网络，即连接用户态的交换机和用户态的控制器。

mininet.topo 模块

维护网络拓扑相关的信息。

除了一些固定结构的拓扑类之外，还提供了 mininet.topolib 模块，提供用户自己创建复杂拓扑相关的库，目前仅包括一个 mininet.topolib.TreeTopo 拓扑，是个树拓扑类，给定深度和广度可以自己生成相应的标准树拓扑。

mininet.topo.MultiGraph

表示一个图结构。 类似于networkx中的图G(V,E)的概念。主要维护节点、边信息。 在MultiGraph中，节点就是一个序号，边则通过节点和节点所对应的连接列表中元素来表示。节点和节点的连接列表的对应关系通过字典结构来维护。

init

```
def __init__( self ):
        self.data = {}
```

图结构最主要的功能就是维护一个字典。Key 是节点，value 是该节点所连接的所有的其他节点的列表。

```
add_node
    def add_node( self, node ):
        "Add node to graph"
        self.data.setdefault( node, [] )
```

添加一个节点，实际上就是添加一个 key 到 data 字典中。

```
add_edge
    def add_edge( self, src, dest ):
        "Add edge to graph"
        src, dest = sorted( ( src, dest ) )
        self.add_node( src )
        self.add_node( dest )
        self.data[ src ].append( dest )
```

添加一条边，实际上就是添加两个节点，然后将连接信息放到 data 字典中（需要注意的是一条边的信息仅被保存了一次，即放到序号较小的节点对应的 list 中）。

mininet.topo.Topo

拓扑基类，默认的拓扑图被 multigraph 类维护，此外还包括节点、连接信息等。主要的方法就是添加节点、连接等。 Topo 中一个 node 实际上就是图结构中的一个节点，一个 port 是全局维护增长的源和目的 node 所对应的序号，而连接

init

```
def __init__(self, hopts=None, sopts=None, lopts=None):
        """Topo object:
           hinfo: default host options
           sopts: default switch options
           lopts: default link options"""
        self.g = MultiGraph()
        self.node_info = {}
        self.link_info = {}  # (src, dst) tuples hash to EdgeInfo objects
        self.hopts = {} if hopts is None else hopts
        self.sopts = {} if sopts is None else sopts
        self.lopts = {} if lopts is None else lopts
        self.ports = {}  # ports[src][dst] is port on src that connects to dst
```

addNode

```
def addNode(self, name, **opts):
        """Add Node to graph.
           name: name
           opts: node options
           returns: node name"""
        self.g.add_node(name)
        self.node_info[name] = opts
        return name
```

添加节点方法被添加主机 addHost、交换机 addSwitch、控制器 addController 等方法使用。

该方法在图上添加一个节点，然后添加对应的节点信息到拓扑的参数上。

addPort

```
def addPort(self, src, dst, sport=None, dport=None):
        '''Generate port mapping for new edge.
        @param src source switch name
        @param dst destination switch name
        '''
        self.ports.setdefault(src, {})
        self.ports.setdefault(dst, {})
        # New port: number of outlinks + base
        src_base = 1 if self.isSwitch(src) else 0
        dst_base = 1 if self.isSwitch(dst) else 0
        if sport is None:
            sport = len(self.ports[src]) + src_base
        if dport is None:
            dport = len(self.ports[dst]) + dst_base
        self.ports[src][dst] = sport
        self.ports[dst][src] = dport
```

addPort 会同时创建源和目标节点上的端口号信息。 拓扑中所有的 port 都被 self.ports 结构维护，其中 ports[src][dst] 表示在 src 节点上的 port，该 port 所在 link 连接到 dst 节点。 添加一个 port 就是更新了这些信息。

addLink

```
def addLink(self, node1, node2, port1=None, port2=None,
                **opts):
        """node1, node2: nodes to link together
           port1, port2: ports (optional)
           opts: link options (optional)
           returns: link info key"""
        if not opts and self.lopts:
            opts = self.lopts
        self.addPort(node1, node2, port1, port2)
        key = tuple(self.sorted([node1, node2]))
        self.link_info[key] = opts
        self.g.add_edge(*key)
        return key
```

添加一条连接实际上就是添加对应的两个 port，并在图上添加上边。

mininet.topo.LinearTopo

表示一个线行拓扑，交换机连接成一条链，每个交换机上挂载相等个数的主机。

mininet.topo.SingleSwitchTopo

单个交换机上挂载若干主机，主机序号按照从小到大的顺序依次挂载到交换机的各个端口上。

mininet.topo.SingleSwitchReversedTopo

单个交换机上挂载若干主机，主机序号按照从大到小的顺序依次挂载到交换机的各个端口上。

mininet.cli 模块

主要包括 CLI 类，该类继承自 Python 库的 Cmd 类。

提供对 CLI 的支持，创建 Mininet 的 bash，接受通过 bash 传输的 Mininet 命令，形成可以进行交互的 Mininet 命令行环境。

主要方法包括初始化之后提供一个界面，通过 Python 库的 Cmd 类的 cmdloop() 方法不断执行输入的命令。这些命令可以是指定对某个节点进行的操作或者是对 Mininet 对象本身。

对于大部分对 Mininet 对象的操作命令 xxx，会调用 Python 库的 Cmd 类的 onecmd() 方法来执行，该方法会对应调用 do_xxx 方法。

各个 do_xxx 方法分别用于执行支持的命令。例如 do_dpctl() 方法响应用户输入 dpctl 相关命令。目前包括 dpctl，dump，EOF，exit，gterm，help，intfs，iperf，iperfudp，link，net，nodes，noecho，pingall，pingallfull，pingpair，pingparifull，px，py，quit，sh，source，time，x，xterm 等。

mininet.clean 模块

提供对执行 Mininet 后的清理工作，主要包括 cleanup() 函数，该函数实际上调用了 sh() 函数。

cleanup() 函数主要包括清除僵尸进程，临时文件，X11 tunnel，额外的内核态 datapath，ovs datapath，ip link 等。

实现过程主要是通过调用 subprocess 模块（主要用于执行外部命令和程序）中的 Popen 类中方法来对进程发送指令。

```
def sh( cmd ):
    "Print a command and send it to the shell"
    info( cmd + '\n' )
    return Popen( [ '/bin/sh', '-c', cmd ], stdout=PIPE ).communicate()[ 0 ]
```

communicate() 是 Popen 对象的一个方法，该方法会阻塞父进程，直到子进程完成。 通过指定 stdout=PIPE，可以通过 stdout 获取程序的返回值。通过列表传入要执行的命令和参数。

mininet.log 模块

利用 logging 包，主要提供进行 log 相关的功能，包括三个类：MininetLogger、Singleton、StreamHandlerNoNewline。

mininet.log.MininetLogger

自定义的 logger 类。 提供输出 log、配置 LogLevel 功能。

mininet.log.Singleton

软件设计模式，限定所创建的类只能有一个实例。供 MininetLogger 使用。

mininet.log.StreamHandlerNoNewline

自动添加换行和对流进行格式化，供 MininetLogger 使用。

mininet.moduledeps 模块

定义几个对 Linux 系统中内核模块进行操作的函数，包括列出模块 lsmod，移除模块 rmmod，探测模块 modprobe 和处理模块的依赖等。

mininet.term 模块

支持 term 相关的命令，例如在主机上创建一个 xterm。实现依赖于 socat 和 xterm。

mininet.util 模块

一些辅助的方法。包括如下重要的方法。

- errFail 利用 errRun（利用 popen 来在 shell 中执行命令）来执行一个命令，并且如果执行不成功则抛出异常。

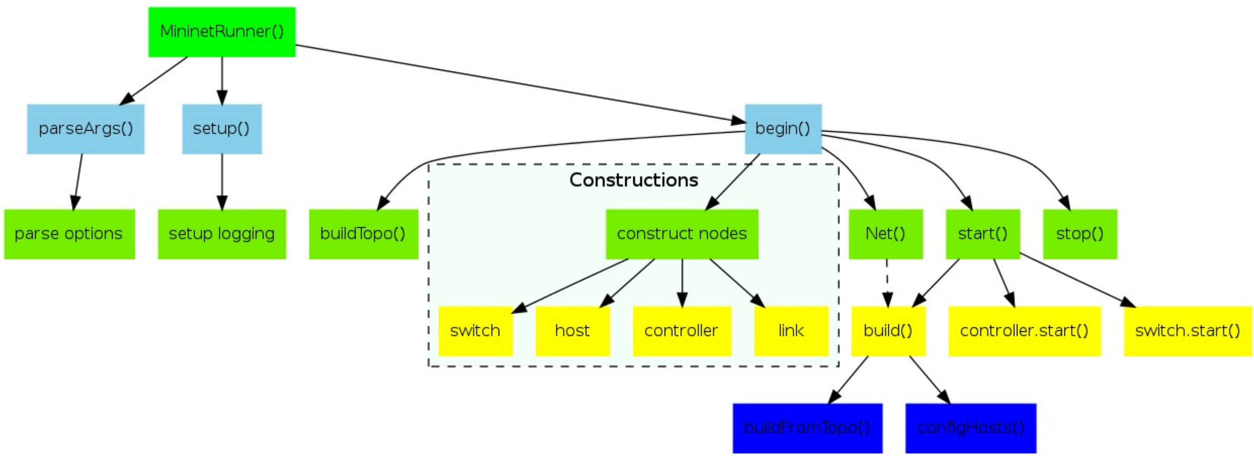
运行代码和示例

mn

该脚本定义了一个 MininetRunner 类，用来表示模拟网络的主程序。

主要过程是创建一个 MininetRunner() 实例，依次解析传入参数，进行初始化后开启网络。

整体过程如下图所示。



其中 Mininet 类的 start() 方法是核心的启动过程，主要包括调用 build 方法来根据拓扑创建网络、控制器、交换机、主机和连接等。之后依次启动控制器和交换机进程。 在执行完 start() 之后，通过 test 参数来判断 mininet 运行的模式。

```
if test == 'none':
            pass
        elif test == 'all':
            mn.start()
            mn.ping()
            mn.iperf()
        elif test == 'cli':
            CLI( mn )
        elif test != 'build':
            getattr( mn, test )()
```

默认情况下，参数为 cli，即进入到控制台模式，允许用户自己输入对 Mininet 的操作命令。 最终执行 mininet.stop() 进行删除资源的工作。

示例程序

Mininet 代码中带有了大量的示例程序，供大家参考和理解代码。所有的示例程序都在 example 目录下，包括

baresshd.py:

使用 Mininet 的中层 API 来在一个 namespace 中创建主机、链路，并在主机上启动 sshd 进程，让用户可以登录。并未使用 OpenFlow。

consoles.py:

为每一个节点都创建一些 console 窗口，并允许用户对这些节点进行操作和观测，支持图形界面。

controllers.py:

使用一个自定义的Switch() 子类，创建一个带有多个控制器的网络。

controllers2.py:

创建一个拥有多个控制器的网络，通过创建空的网络，添加节点和手动启动交换机实现。

controlnet.py:

通过创建两个mininet对象来建模一个控制网络和数据网络。

cpu.py:

在不同的 CPU 限制下测试 iperf 的带宽性能。

emptynet.py:

演示创建一个空的网络，之后添加节点进去。

hwintf.py:

添加一个接口（例如一个物理接口）到一个网络中。

limit.py:

演示如何使用 link 和 CPU 限制。

linearbandwidth.py:

基于 Topo 创建一个拓扑子类，并进行简单的测试。

miniedit.py:

通过一个图形界面的编辑器来创建网络。

multiping.py:

使用 node.monitor() 来检测多个主机的输出。

multipoll.py:

检测多个主机的输出文件。

multitest.py:

创建一个网络，并在其上进行多个测试。

nat.py:

将 Mininet 的网络通过 nat 连接到外部网络中。

popen.py:

使用 host.popen() 和 pmonitor() 来检测多个主机。

popenpoll.py:

使用 node.popen() 和 pmonitor() 检测多个主机的输出。

scratchnet.py, scratchnetuser.py:

使用底层的 Mininet 函数来创建网络。

simpleperf.py:

配置网络和 CPU、带宽限制等。

sshd.py:

在每个主机里面运行一个 sshd 进程，使得用户可以通过 ssh 来访问主机。这需要将 Mininet 的数据网络连接到 root 名字空间的一个接口上。一般的，控制网络已经在 root 名字空间了，所以默认已经被连接。

tree1024.py:

创建一个 1024 主机的网络，然后运行 CLI。根据系统资源情况，可能需要利用 sysctl 进行相关修改。

treeping64.py:

创建一个 64 主机的树状网络，利用 ping 来检查连通性。